An Incremental Algorithm for Transition-based CCG Parsing

An Incremental Algorithm for Transition-based CCG Parsing

Bharat Ram Ambati[1] , Tejaswini Deoskar[1] , Mark Johnson[2] , Mark Steedman[1]
[1] ILCC, School of Informatics, University of Edinburgh
[2] Department of Computing, Macquarie University
bharat.ambati@ed.ac.uk, mark.johnson@mq.edu.au, {tdeoskar,steedman}@inf.ed.ac.uk

Abstract

Incremental parsers have potential advantages for applications like language modeling for machine translation and speech recognition. We describe a new algorithm for incremental transition-based Combinatory Categorial Grammar parsing. As English CCGbank derivations are mostly right branching and non-incremental, we design our algorithm based on the dependencies resolved rather than the derivation. We introduce two new actions in the shift-reduce paradigm based on the idea of 'revealing' (Pareschi and Steedman, 1987) the required information during parsing. On the standard CCGbank test data, our algorithm achieved improvements of 0.88% in labeled and 2.0% in unlabeled F-score over a greedy non-incremental shift-reduce parser.

1 Introduction

Combinatory Categorial Grammar (CCG) (Steedman, 2000) is an efficiently parseable, yet linguistically expressive grammar formalism. In addition to predicate-argument structure, CCG elegantly captures the unbounded dependencies found in grammatical constructions like relativization, coordination etc. Availability of the English CCGbank (Hockenmaier and Steedman, 2007) has enabled the creation of several robust and accurate wide-coverage CCG parsers (Hockenmaier and Steedman, 2002; Clark and Curran, 2007; Zhang and Clark, 2011). While the majority of CCG parsers use chart-based approaches (Hockenmaier and Steedman, 2002; Clark and Curran, 2007), there has been some work on developing shift-reduce parsers for CCG (Zhang and Clark, 2011; Xu et al., 2014). Most of these parsers model normal-form CCG derivations (Eisner, 1996), which are mostly right-branching trees : hence are not incremental in nature. The dependency models of Clark and Curran (2007) and Xu et al. (2014) model dependencies rather than derivations, but do not guarantee incremental analyses.

Besides being cognitively plausible (Marslen-Wilson, 1973), incremental parsing is more useful than non-incremental parsing for some applications. For example, an incremental analysis is required for integrating syntactic and semantic information into language modeling for statistical machine translation (SMT) and automatic speech recognition (ASR) (Roark, 2001; Wang and Harper, 2003).

This paper develops a new incremental shift-reduce algorithm for parsing CCG by building a dependency graph in addition to the CCG derivation as a representation. The dependencies in the graph are extracted from the CCG derivation. A node can have multiple parents, and hence we construct a dependency graph rather than a tree. Two new actions are introduced in the shift-reduce paradigm for "revealing" (Pareschi and Steedman, 1987) unbuilt structure during parsing. We build the dependency graph in parallel to the incremental CCG derivation and use this graph for revealing, via these two new actions. On the standard CCGbank test data, our algorithm achieves improvements of 0.88% in labeled F-score and 2.0% in unlabeled F-score over a greedy non-incremental shift-reduce algorithm. As our algorithm does not model derivations, but rather models transitions, we do not need a treebank

Human Language Technologies: The 2015 Annual Conference of the North American Chapter of the ACL, pages 53–63,
Denver, Colorado, May 31 – June 5, 2015.

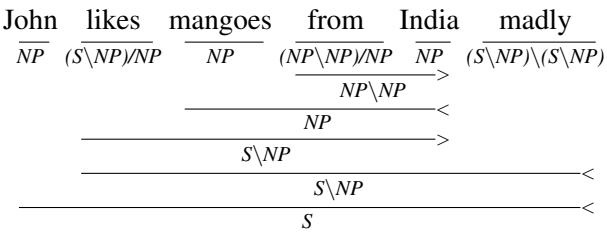

Figure 1: Normal form CCG derivation.

of incremental CCG derivations and can train on the dependencies in the existing treebank. Our approach can therefore be adapted to other languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005; Ambati et al., 2013).

The rest of the paper is arranged as follows. Section 2 gives a brief introduction to related work in the areas of CCG parsing and incremental parsing. In section 3, we describe our incremental shift-reduce parsing algorithm. Details about the experiments, evaluation metrices and analysis of the results are in section 4. We conclude with possible future directions in section 5.

2 Related Work

In this section, we first give a brief introduction to various available CCG parsers. Then we describe approaches towards incremental and greedy parsing.

2.1 CCG Parsers

There has been a significant amount of work on developing chart-based parsers for CCG. Both generative (Hockenmaier and Steedman, 2002) and discriminative (Clark et al., 2002; Clark and Curran, 2007; Auli and Lopez, 2011; Lewis and Steedman, 2014) models have been developed. As these parsers employ a bottom-up chart-parsing strategy and use normal-form CCGbank derivations which are right-branching, they are not incremental in nature. In an SVO (Subject-Verb-Object) language, these parsers first attach the object to the verb and then the subject.

Two major works in shift-reduce CCG parsing with accuracies competitive with the widely used Clark and Curran (2007) parser (C&C) are Zhang and Clark (2011) and Xu et al. (2014). Zhang and Clark (2011) used a global linear model trained discriminatively with the averaged perceptron (Collins, 2002) and beam search for their shift-reduce CCG parser. Xu et al. (2014) developed a dependency model for shift-reduce CCG parsing using a dynamic oracle technique. Unlike the chart parsers, both these parsers can produce fragmentary analyses when a complete spanning analysis is not found. Both these shift-reduce parsers are more incremental than standard chart based parsers. But, as they employ an arc-standard (Yamada and Matsumoto, 2003) shift-reduce strategy on CCG-bank, given an SVO language, these parsers are not guaranteed to attach the subject before the object.

2.2 Incremental Parsers

A strictly incremental parser is one which computes the relationship between words as soon as they are encountered in the input. Shift-reduce CCG parsers rely either on CCGbank derivations (Zhang and Clark, 2011) which are non-incremental, or on dependencies (Xu et al., 2014) which could be incremental in simple cases, but do not guarantee incrementality. Hassan et al. (2009) developed a semi-incremental CCG parser by transforming the English CCGbank into left branching derivation trees. The strictly incremental version performed with very low accuracy but a semi-incremental version gave a balance between incrementality and accuracy. There is also some work on incremental parsing using grammar formalisms other than CCG like phrase structure grammar (Collins and Roark, 2004) and tree substitution grammar (Sangati and Keller, 2013).

2.3 Greedy Parsers

There has been a significant amount of work on greedy shift-reduce dependency parsing. The Malt parser (Nivre et al., 2007) is one of the earliest parsers based on this paradigm. Goldberg and Nivre (2012) improved learning for greedy parsers by using dynamic oracles rather than a single static transition sequence as the oracle. In all the standard shift-reduce parsers, when two trees combine, only the top node (root) of each tree participates in the action. Sartorio et al. (2013) introduced a technique where in addition to the root node, nodes on the right and left periphery respectively are also available for attachment in the parsing process. A non-monotonic parsing strategy was introduced by Honnibal et al. (2013), where an action taken during the parsing process is revised based on future context.

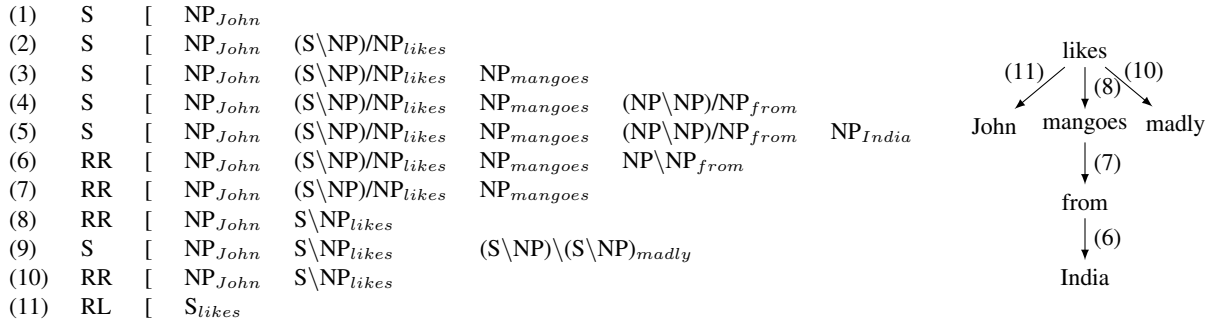

Figure 2: NonInc - Sequence of actions with parser configuration and the corresponding dependency graph.

Though the performance of these greedy parsers is less accurate than related parsers using a beam (Zhang and Nivre, 2011), greedy parsers are interesting as they are very fast and are practically useful in large-scale applications such as parsing the web and online machine translation or speech recognition. In this work, we develop a new greedy transition-based algorithm for incremental CCG parsing, which is more incremental than Zhang and Clark (2011) and Xu et al. (2014) and more accurate than Hassan et al. (2009). Our algorithm is not strictly incremental as we only produce derivations which are compatible with the Strict Competence Hypothesis (Steedman, 2000) (details in §3.2.3).

3 Algorithms

We first describe the Zhang and Clark (2011) style shift-reduce algorithm for CCG parsing. Then we explain our incremental algorithm based on the "revealing" technique for shift-reduce CCG parsing.

3.1 Non Incremental Algorithm (NonInc)

This is our baseline algorithm and is similar to Zhang and Clark (2011)'s algorithm (henceforth NonInc). It consists of an input buffer and a stack and has four major parsing actions.

- Shift - X `(S)` : Pushes a word from the input buffer to the stack and assigns a CCG category X. This action performs category disambiguation as well, as X can be any of the categories assigned by a supertagger.
- Reduce Left - X `(RL)` : Pops the top two nodes from the stack, combines them into a new node and pushes it back onto the stack with a category X. This corresponds to binary rules in the CCGbank (e.g. CCG combinators like function application, composition etc., and punctuation rules). In this action the right node is the head and hence the left node is reduced.
- Reduce Right - X `(RR)` : This action is similar to the RL (Reduce Left -X) action, except that in this action the right node is reduced since the left node is the head.
- Unary - X `(U)` : Pops the top node from the stack, converts it into a new node with category X and pushes it back on the stack. The head remains the same in this action. This action corresponds to unary rules in the CCGbank (unary type-changing and type-raising rules).

Figure 1 shows a normal-form CCG derivation for an example sentence 'John likes mangoes from India madly'. Figure 2 shows the sequence of steps using the NonInc algorithm for parsing the sentence. For simplicity and space reasons, unary productions leading to `NP` are not described. From step 1 through step 5, the first five words in the sentence (John, likes, mangoes, from, India) are shifted with corresponding categories using shift actions (S). In step 6, `(NP\NP)/NP:from` and `NP:India` are combined using the Reduce-Right (RR) action to form `NP\NP:from` which is combined with `NP:mangoes` in step 7 to form `NP:mangoes`. Step 8 combines `(S\NP)/NP:likes` with `NP:mangoes` to form `S\NP:likes` using RR action. Then the next word 'madly' is shifted in step 9, which is then combined with `S\NP:likes` in step 10. In step 11, `NP:John` and `S\NP:likes` are combined using Reduce-Left (RL) action leading to `S:likes`. The parsing process terminates at this step as there are no more tokens in the input buffer and as there is only a single node left in the stack.

(1)	S	[	NP_{John}		
(2)	S	[	NP_{John}	$(\text{S}\backslash\text{NP})/\text{NP}_{likes}$	
(3)	RL	[	S/NP_{likes}		
(4)	S	[	S/NP_{likes}	$\text{NP}_{mangoes}$	
(5)	RR	[	S_{likes}		
(6)	S	[	S_{likes}	$(\text{NP}\backslash\text{NP})/\text{NP}_{from}$	
(7)	S	[	S_{likes}	$(\text{NP}\backslash\text{NP})/\text{NP}_{from}$	NP_{India}
(8)	RR	[	S_{likes}	$\text{NP}\backslash\text{NP}_{from}$	
(9)	RRev	[	S_{likes}		
(10)	S	[	S_{likes}	$(\text{S}\backslash\text{NP})\backslash(\text{S}\backslash\text{NP})_{madly}$	
(11)	LRev	[	S_{likes}		

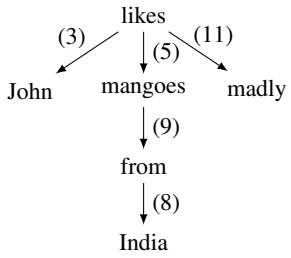

Figure 3: RevInc - Sequence of actions with parser configuration and the corresponding dependency graph.

We use indexed CCG categories (Clark et al., 2002) and obtain the CCG dependencies after every action to build the dependency graph in parallel to the CCG derivation. This is similar to Xu et al. (2014) but differs from Zhang and Clark (2011), who extract the dependencies at the end after obtaining a derivation for the entire sentence. Figure 2 also shows the dependency graph generated and the arc labels give the step ID after which the dependency is generated.

3.2 Revealing based Incremental Algorithm (RevInc)

The NonInc algorithm described above is not incremental because it relies purely on the mostly right-branching CCG derivation. In our example sentence, the verb (likes) combines with the subject (John) only at the end (step ID = 11) after all the remaining words in the sentence are processed, making the parse non-incremental. In this section we describe a new incremental algorithm based on a 'revealing' technique (Pareschi and Steedman, 1987) which tries to build the most incremental derivation.

3.2.1 Revealing

Pareschi and Steedman (1987)'s original version of revealing was defined in terms of (implicitly higher-order) unification. It was based on the following observation. If we think of categories as terms in a logic programming language, then while we usually think of CCG combinatory rules like the following as applying with the two categories on the left X/Y and Y as inputs, say instantiated as S/NP and NP, to define the category X on the right as S, in fact instantiating *any* two of those categories defines the third.

$$X/Y \;\; Y \implies X$$

For example, if we define X and X/Y as S and S/NP, we clearly define Y as NP. They proposed to use unification-based revealing to recover unbuilt constituents in from the result of overly-greedy incremental parsing. A related second-order matching-based mechanism was used by (Kwiatkowski et al., 2010) to decompose logical forms for semantic parser induction.

The present incremental parser uses a related revealing technique confined to the right periphery. Using CCG combinators and rules like type-raising followed by forward composition, we combine nodes in the stack if there is a dependency between them. However, this can create problems for the newly shifted node as its dependent might already have been reduced. For instance, if the object 'mangoes' is reduced after it is shifted to the stack, then it won't be available for the preposition phrase (PP) 'from India' (of course, this goes for more complex NPs as well). We have to extract 'mangoes', which is hidden in the derivation, so as to make the correct attachment to the PP. This is where revealing comes into play. Mangoes is 'revealed' so that it is available to attach to the PP following it, although it has already been reduced. To handle this, in addition to the four actions of the NonInc algorithm, we introduce two new actions: Left Reveal (`LRev`) and Right Reveal (`RRev`). For this, after every action, in addition to updating the stack we also keep track of the dependencies resolved and update the dependency graph accordingly[1]. In other words, we build the dependency graph for the

[1]Xu et al. (2014) also obtain CCG dependencies after every action. But they don't have a dependency graph which is updated based on the CCG derivation and used in the CCG parsing (in our case for LRev and RRev actions).

(a) RRev (b) LRev

Figure 4: RRev and LRev actions.

sentence in parallel to the CCG derivation. As these dependencies are extracted from the CCG derivation, a node can have multiple parents and hence we construct a dependency graph rather than a tree.

- Left Reveal (`LRev`) : Pop the top two nodes in the stack (left, right). Identify the left node's child with a subject dependency. Abstract over this child node and split the category of left node into two categories. Combine the nodes using CCG combinators accordingly. VP modifiers like VP coordination require this action.
- Right Reveal (`RRev`) : Pop the top two nodes in the stack (left, right). Check the right periphery of the left node in the dependency graph, extract all the nodes with compatible CCG categories and identify all the possible nodes that the right node can combine with. Abstract over this node (e.g. object), split the category into two categories accordingly and combine the nodes using CCG combinators. Constructions like NP coordination, and PP attachment require this action.

3.2.2 Worked Example

Figure 3 shows the sequence of steps for the example sentence described above. In steps 1 and 2, the first two words in the sentence: 'John' and 'likes', are shifted from the input buffer to the stack. In addition to standard CCG combinators of application and composition, we also use type-raising followed by forward composition[2]. In step 3, the category of the left node 'John', `NP`, is type-raised to `S/(S\NP)` which is then combined with the category of right node 'likes', `(S\NP)/NP`, using forward composition operator to yield the category `S/NP`. This step also updates the dependency graph with an edge between 'John' and 'likes', where 'likes' is the parent and 'John' is the child. The next word 'mangoes' is shifted in step 4 and combined with `S/NP:likes` in step 5 using RR action yielding `S:likes`. After this step, the dependency graph will have 'likes' as the root, with 'John' and 'mangoes' as its children. In this way, as our algorithm tries to be more incremental, both subject and object arguments are resolved as soon as the corresponding tokens are shifted to the stack.

In steps 6 and 7, the next two words 'from' and 'India' are shifted to the stack. Step 8 combines `(NP\NP)/NP:from` and `NP:India` using RR action to form `NP\NP:from`. Now, we apply the `RRev` action in step 9 to correctly attach 'from' to 'mangoes'. In `RRev` we first check the right periphery and identify a possible node to be attached, 'mangoes', which is the object argument of the verb 'likes'. We abstract over this object and split the category in the following manner: If `X` is the category of the left node and `Y\Y` is the category of the right node, then `X` is split into `X/Y` and `Y` with corresponding heads. The head of the left node will be the head of `X/Y`, and the dependency graph helps in identifying the correct head for `Y`. Now, `Y` and `Y\Y` can be combined using the backward application rule to form `Y`, which can be combined with `X/Y` to form `X` back. In our example sentence, `S:likes` is split into `S/NP:likes` and `NP:mangoes`. `NP:mangoes` is combined with `NP\NP:from` to form `NP:mangoes`, which in return combines with `S/NP:likes` and forms back `S:likes`. Figure 4(a) sketches this process. This action also updates the dependency graph with a dependency between 'mangoes' and 'from'.

The next word 'madly' is shifted in step 10, after which the stack has two nodes `S:likes` and `(S\NP)\(S\NP):madly`. We apply the `LRev` action to combine these two nodes. We abstract over the subject of the left node, 'likes', and split the category. Here, `S:likes` is split into `NP:John` and `S\NP:likes`. `S\NP:likes` is combined with `(S\NP)\(S\NP):madly` to form `S\NP:likes`,

[2]Type-raising followed by forward composition is treated as a single step. Without this, after type-raising, the parser has to check all possible actions before applying forward composition, making it slower.

which in return combines with NP:John and forms back S:likes. The dependency graph is updated with a dependency between 'likes' and 'madly'. Note that the final output is a standard CCG tree. Figure 4(b) shows this LRev action.

3.2.3 Analysis

Our incremental algorithm uses a combination of the CCG derivation and a dependency graph that helps to 'reveal' unbuilt structure in the CCG derivation by identifying heads of the revealed categories. For example in Figure-4a, in RRev action, S:likes is split into S/NP:likes and NP:mangoes. The splitting of categories is deterministic but the right periphery of the dependency graph helps in identifying the head, which is 'mangoes'. The theoretical idea of 'revealing' is from Pareschi and Steedman (1987), but they used only a toy grammar without a model or empirical results. Checking the right periphery is similar to Sartorio et al. (2013) and abstracting over the left or right argument is similar to Dalrymple et al. (1991). Currently, we abstract only over arguments. Adding a new action to abstract over the verb as well will make our algorithm handle ellipses in the sentences like 'John likes mangoes and Mary too' similar to Dalrymple et al. (1991) but we leave that for future work.

Our system is monotonic in the sense that the set of dependency relationships grows monotonically during the parsing process. Our algorithm gives derivations almost as incremental as Hassan et al. (2009) but without changing the lexical categories and without backtracking. The only change we made to the CCGbank is making the main verb the head of the auxiliary rather than the reverse as in CCGbank derivations. In the right derivational trees of CCGbank, the main verb is the head for its right side arguments and the auxiliary verb is the head for the left side arguments in the derivation. Not changing the head rule would make our algorithm use the costly reveal actions significantly more, which we avoid by changing the head direction. 3% of the total dependencies are affected by this modification.

Though our algorithm can be completely incremental, we currently compromise incrementality in the following cases:

(a) no dependency between the nodes in the stack
(b) unary type-changing and non-standard binary rules
(c) adjuncts like VP modifiers and coordinate constructions like VP, sentential coordination.

We find empirically that extending incrementality to cover these cases actually reduces parsing performance significantly. It also violates the Strict Competence Hypothesis (SCH) (Steedman, 2000), which argues on evolutionary and developmental grounds that the parser can only build constituents that are typable by the competence grammar. We explored the adjunct case of attaching only the preposition first rather than creating a complete prepositional phrase and then attaching it to correct parent. In our example sentence, this would be the case of attaching the preposition 'from' to its parent using RRev and then combining the NP 'India' accordingly as opposed to creating the preposition phrase 'from India' first and then using RRev action to attach it to the correct parent. Though the former is more incremental, it is inconsistent with the SCH. The latter analysis is consistent with strict competence and also gave better parsing performance while compromising incrementality only slightly. The empirical impact of these differing degrees of incrementality on extrinsic evaluation of our algorithm in terms of language modeling for SMT or ASR is left for future work.

Using our incremental algorithm, we converted the CCGbank derivations into a sequence of shift-reduce actions. We could convert around 98% of the derivations, which is the coverage of our algorithm, recovering around 99% dependencies. Problematic cases are mainly the ones which involve non-standard binary rules, and punctuations with lexical CCG categories other than 'conj', used as a conjunction, or ',' which is treated as a punctuation mark.

4 Experiments and Results

We re-implemented Zhang and Clark (2011)'s model for our experiments. We used their global linear model trained with the averaged perceptron (Collins, 2002). We applied the early-update strategy of Collins and Roark (2004) while training. In this strategy, when we don't use a beam, decoding is stopped when the predicted action is different from the gold action and weights are updated accordingly. We use the feature set of Zhang and Clark (2011) (Z&C) for the NonInc algorithm. This feature set comprises of features over the top four nodes in the

stack and the next four words in the input buffer. Complete details of the feature set can be found in their paper. For our own model, RevInc, in addition to these features used for NonInc, we also provide features based on the right periphery of top node in the stack. For nodes in the right periphery, we provide uni-gram and bi-gram features based on the node's CCG category. For example, if S0 is the node on the top of the stack, B1 is the bottom most node in the right periphery, and c represent the node's CCG category, then B1c, and B1cS0c are the uni-gram and bi-gram features respectively.

Unlike Z&C, we do not use a beam for our experiments, although we use a beam of 16 for comparison of our results with their parser. The latter gives competitive results with the state-of-the-art CCG parsers. Z&C and Xu et al. (2014), use C&C's `generate` script and unification mechanism respectively to extract dependencies for evaluation. C&C's grammar doesn't cover all the lexical categories and binary rules in the CCGbank. To avoid this, we adapted Hockenmaier's scripts used for extracting dependencies from the CCGbank derivations. These are the two major divergences in our re-implementation from Z&C.

4.1 Data and Settings

We use standard CCGbank training (sections 02 − 21), development (section 00) and testing (section 23) splits for our experiments. All sentences in the training set are used to train NonInc. But for RevInc, we used 98% of the training set (the coverage of our algorithm). We use automatic POS-tags and lexical CCG categories assigned using the C&C POS tagger and supertagger respectively for development and test data. For training data, these tags are assigned using ten-way jackknifing. Also, for lexical CCG categories, we use a multitagger which assigns k-best supertags to a word rather than 1-best supertagging (Clark and Curran, 2004). The number of supertags assigned to a word depends on a β parameter. Unlike Z&C, the default value of β gave us better results rather than decreasing the value. Not using a beam could be the reason for this.

Following Z&C and Xu et al. (2014), during training, we also provide the gold CCG lexical category to the list of CCG lexical categories for a word if it is not assigned by the supertagger.

4.2 Connectedness and Waiting Time

Before evaluating the performance of our algorithm, we introduce two measures of incrementality: connectedness and waiting time. In a shift-reduce parser, a derivation is fully connected when all the nodes in the stack are connected leading to only one node in the stack at any point of time. We measure the average number of nodes in the stack before shifting a new token from input buffer to the stack, which we call as connectedness. For a fully connected incremental parser like Hassan et al. (2009), connectedness would be one. As our RevInc algorithm is not fully connected, this number will be greater than one. For example, in a noun phrase 'the big book', when 'the' and 'big' are in the stack, as there is no dependency between these two words, our algorithm doesn't combine these two nodes resulting in having two nodes in the stack[3]. Second column in Table 1 gives this number for both NonInc and RevInc algorithms. Though our algorithm is not fully connected, connectedness of our algorithm is significantly lower than the NonInc algorithm as our algorithm is more incremental.

Algorithm	*Connectedness*	*Waiting Time*
NonInc	4.62	2.98
RevInc	**2.15**	**0.69**

Table 1: Connectedness and waiting time.

We define waiting time as the number of nodes that need to be shifted to the stack before a dependency between any two nodes in the stack is resolved. In our example sentence, there is a dependency between 'John' and 'likes'. For NonInc, this dependency is resolved only after all the four remaining words in the sentence are shifted. In other words, it has to wait for four more words before this dependency is resolved and hence the waiting time is four. Whereas, in our RevInc algorithm, this dependency is resolved immediately, without waiting for more words to be shifted, and hence the waiting time is zero. The third column in Table 1 gives the waiting time for both the algorithms. Since we compromised incrementality in cases like coordination, waiting time for our RevInc algorithm is not zero but it is significantly lower than the

[3]This is a case where the dependencies are not true to the CCG grammar, and make our algorithm less incremental than SCH would allow.

Algorithm	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
NonInc (beam=1)	**92.57**	82.60	87.30	**85.12**	75.96	80.28	91.10
RevInc (beam=1)	91.62	**85.94**	**88.69**	83.42	**78.25**	**80.75**	90.87
NonInc (beam=16)	92.71	89.66	91.16	85.78	82.96	84.35	92.51
Z&C (beam=16)*	-	-	-	87.15	82.95	85.00	92.77

Table 2: Performance on the development data. *: These results are from the Z&C paper.

NonInc algorithm and hence more incremental. This property is likely to be crucial for future applications in ASR and SMT language modeling.

4.3 Results and Analysis

We trained the perceptron for both NonInc and RevInc algorithms using the CCGbank training data for 30 iterations, and the models which gave best results on development data are directly used for test data. Table 2 gives the unlabeled precision (UP), recall (UR), F-score (UF) and labeled precision (LP), recall (LR), F-score (LF) results of both NonInc and RevInc approaches on the development data. Last column in the table gives the category accuracy. We used the modified CCGbank for all experiments, including NonInc, for consistent comparisons. For NonInc, the modification decreased unlabeled F-score by 0.45%, without a major difference in labeled F-score.

Our incremental algorithm gives 1.39% and 0.47% improvements over the NonInc algorithm in unlabeled and labeled F-scores respectively. For both unlabeled and labeled scores, precision of RevInc is slightly lower than NonInc but the recall of RevInc is much higher than NonInc resulting in a better F-score for RevInc. As NonInc is not incremental and as it uses more context to the right while making a decision, it makes more precise actions. But, on the other hand, if a node is reduced, it is not available for future actions. This is not a problem for our RevInc algorithm which is the reason for higher recall. For example, in the example sentence, 'John likes mangoes from India madly', if the object 'mangoes' is reduced after it got shifted to the stack, then in case of NonInc, the preposition phrase 'from India' can never be attached to 'mangoes'. But, RevInc makes the correct attachment using `RRev` action. Category accuracy of NonInc is better than RevInc, since NonInc can use more context before taking a complex action and is less prone to error propagation compared to RevInc.

To compare these results in the perspective of Z&C's parser we also trained our NonInc parser with a beam size of 16 similar to Z&C. The second last row in Table 2 (NonInc (beam=16)) shows these results and the last row presents the results from their paper. Results with our implementation of Z&C are 0.65% lower than the published results, possibly due to the modification made in the head rule, and other minor differences like the supertagger beta value. Unlabeled and labeled F-scores of our RevInc parser are lower than these numbers. But, given that our RevInc parser doesn't use any beam, these margins are reasonable.

We also analyzed the label-wise scores of both NonInc and RevInc. In general, NonInc is better in precision and RevInc is better in recall. In the case of verbal arguments (`(S\NP)/NP`) and verbal modifiers (`(S\NP)\(S\NP)`), the F-score of RevInc is better than that of NonInc. But NonInc performed better than RevInc in the case of preposition phrase (PP) attachments (`(NP\NP)/NP`, `((S\NP)\(S\NP))/NP`). More context is required for better PP attachment which is provided by the fact that NonInc has a context of several unreduced types for the model to work with, whereas RevInc has fewer. Whereas actions like `LRev` are required to correctly attach the verbal modifiers ('madly') if the subject argument ('John') of the verb ('likes') is reduced early. Table 3 gives the results of these CCG lexical categories.

Category	*RevInc*	*NonInc*
(NP\NP)/**NP**	81.36	**83.21**
(NP**NP**)/NP	78.66	**82.94**
((S\NP)\(S\NP))/**NP**	65.09	**66.98**
((S\NP)\(**S\NP**))/NP	62.69	**65.89**
((S[dcl]\NP)/**NP**	**78.96**	78.29
((S[dcl]**NP**)/NP	**76.71**	75.22
(S\NP)\(**S\NP**)	**80.49**	76.90

Table 3: Label-wise F-score of RevInc and NonInc parsers (both with beam=1). Argument slots in the relation are in bold.

Algorithm	*UP*	*UR*	*UF*	*LP*	*LR*	*LF*	*Cat Acc.*
NonInc (beam=1)	**92.45**	82.16	87.00	**85.59**	76.06	80.55	91.39
RevInc (beam=1)	91.83	**86.35**	**89.00**	84.02	**79.00**	**81.43**	91.17
NonInc (beam=16)	92.68	89.57	91.10	86.20	83.32	84.74	92.70
Z&C (beam=16)*	-	-	-	87.43	83.61	85.48	93.12
Hassan et al. 09*	-	-	86.31	-	-	-	-

Table 4: Performance on the test data. *: These results are from their paper.

We also analyzed the performance of the greedy (beam=1) NonInc and RevInc parsers in terms of parsing speed (excluding pos tagger and supertagger time). NonInc and RevInc parse 110 and 125 sentences/second respectively. Despite the complexity of the revealing actions, RevInc is faster than the NonInc. Significant amount of parsing time is spent on the feature extraction step. Features from top four nodes in the stack and their children are extracted for both the algorithms. Since the average connectedness of RevInc and NonInc are 4.62 and 2.15 respectively, on average, all four nodes in the stack are processed for NonInc and only two nodes are processed for RevInc. Because of this there is significant reduction in the feature extraction step for RevInc compared to NonInc. Also, the complex LRev and RRev actions only constituted 5% of the total actions in the parsing process.

Table 4 presents the results of our approaches on test data. Our incremental algorithm, RevInc, gives 2.0% and 0.88% improvements over NonInc in unlabeled and labeled F-scores respectively on the test data. Results of RevInc without a beam are reasonably close to the results of Z&C which uses a beam of 16. We compare our results with Incremental+Lookahead model of Hassan et al. (2009). They reported 86.31% unlabeled F-score on test data which is 2.69% lower. Note that these F-scores are not directly comparable since Hassan et al. (2009) use simplified lexicalized CCG categories. Our evaluation is based on CCG dependencies which are different from dependencies in the dependency grammar. Hence, we can't directly compare our results with dependency parsers like Zhang and Nivre (2011) and Honnibal et al. (2013).

5 Conclusion and Future Plan

We have designed and implemented a new incremental shift-reduce algorithm based on a version of revealing for parsing CCG (Pareschi and Steedman, 1987). On the standard CCGbank test data, our algorithm achieved improvements of 0.88% and 2.0% in labeled and unlabeled F-scores respectively over the baseline non-incremental shift-reduce algorithm. We achieved this without changing any CCG lexical categories and only changing a single head rule of making the main verb rather than the auxiliary verb the head. Our algorithm models transitions rather than incremental derivations, and hence we don't need an incremental CCGbank. Our approach can therefore be adapted to languages with dependency treebanks, since CCG lexical categories can be easily extracted from dependency treebanks (Cakici, 2005; Ambati et al., 2013). We also designed new measures of incrementality and showed that our algorithm is more incremental than the standard shift-reduce CCG parsing algorithm.

We expect to improve our current model in a number of ways. Providing information about lexical category probabilities (Auli and Lopez, 2011) assigned by the supertagger can be useful during parsing. We would like to explore the limited use of a beam to handle lexical ambiguity by only keeping analyses derived from distinct lexical categories in the beam. Following Xu et al. (2014), we also plan to explore a dynamic oracle strategy. Ultimately, we intend to evaluate the impact of our incremental parser extrinsically in terms of language modeling for SMT or ASR.

Acknowledgments

We thank Mike Lewis, Greg Coppola, Francesco Sartorio and Siva Reddy for helpful discussions. We also thank the three anonymous reviewers for their useful suggestions. This work was supported by ERC Advanced Fellowship 249520 GRAMPLUS, EU IST Cognitive Systems IP Xperience and ARC Discovery grant DP 110102506.

References

Bharat Ram Ambati, Tejaswini Deoskar, and Mark Steedman. 2013. Using CCG categories to improve Hindi dependency parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 604–609, Sofia, Bulgaria.

Michael Auli and Adam Lopez. 2011. A Comparison of Loopy Belief Propagation and Dual Decomposition for Integrated CCG Supertagging and Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 470–480, Portland, Oregon, USA, June.

Ruken Cakici. 2005. Automatic induction of a CCG grammar for Turkish. In *Proceedings of the ACL Student Research Workshop*, pages 73–78, Ann Arbor, Michigan.

Stephen Clark and James R. Curran. 2004. The importance of supertagging for wide-coverage CCG parsing. In *Proceedings of COLING-04*, pages 282–288.

Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building Deep Dependency Structures using a Wide-Coverage CCG Parser. In *ACL*, pages 327–334.

Michael Collins and Brian Roark. 2004. Incremental Parsing with the Perceptron Algorithm. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pages 111–118, Barcelona, Spain, July.

Michael Collins. 2002. Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *Proceedings of the conference on Empirical methods in natural language processing*, EMNLP '02, pages 1–8.

Mary Dalrymple, Stuart M Shieber, and Fernando CN Pereira. 1991. Ellipsis and higher-order unification. *Linguistics and philosophy*, 14(4):399–452.

Jason Eisner. 1996. Efficient Normal-Form Parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 79–86, Santa Cruz, California, USA, June.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India, December.

Hany Hassan, Khalil Sima'an, and Andy Way. 2009. Lexicalized Semi-Incremental Dependency Parsing. In *Proceedings of the Recent Advances in NLP (RANLP'09)*, Borovets, Bulgaria.

Julia Hockenmaier and Mark Steedman. 2002. Generative models for statistical parsing with Combinatory Categorial Grammar. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 335–342, Philadelphia, Pennsylvania.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. 2013. A Non-Monotonic Arc-Eager Transition System for Dependency Parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria, August.

Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. 2010. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1223–1233, Cambridge, MA, October.

Mike Lewis and Mark Steedman. 2014. A* CCG Parsing with a Supertag-factored Model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, Doha, Qatar, October.

W. Marslen-Wilson. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature.*, 244:522–533.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Remo Pareschi and Mark Steedman. 1987. A lazy way to chart-parse with categorial grammars. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, pages 81–88, Stanford, California, USA, July.

Brian Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27:249–276.

Federico Sangati and Frank Keller. 2013. Incremental Tree Substitution Grammar for Parsing and Sentence Prediction. In *Transactions of the Association for Computational Linguistics (TACL)*.

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A Transition-Based Dependency Parser Using a Dynamic Parsing Strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144, Sofia, Bulgaria, August.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press, Cambridge, MA, USA.

Wen Wang and Mary Harper. 2003. Language modeling using a statistical dependency grammar parser. In *Proceedings of the International Workshop on Automatic Speech Recognition and Understanding*, US Virgin Islands.

Wenduan Xu, Stephen Clark, and Yue Zhang. 2014. Shift-Reduce CCG Parsing with a Dependency Model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 218–227, Baltimore, Maryland, June.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical Dependency Analysis with Support Vector Machines. In *In Proceedings of IWPT*, pages 195–206.

Yue Zhang and Stephen Clark. 2011. Shift-Reduce CCG Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 683–692, Portland, Oregon, USA, June.

Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA.